

```

/*****
 * Configuration items in bc7215_lib_config.h
 *
 * TX_HW_FLOW_CONTROL:
 * Refers to flow control for serial data transmission. If the
 * system supports it (e.g., PC and many MCUs like STM32), it
 * is recommended to use hardware flow control combined with a
 * TX FIFO. This allows for non-blocking background transmission
 * where the program simply writes data to the send buffer.
 * Otherwise, without flow control, the driver library must
 * check the state of the BUSY pin before sending each byte
 * during serial transmission. Since infrared signal
 * transmission can take a long time (approx. tens to hundreds
 * of ms), this may cause program blocking.
 * If there is no hardware flow control for serial TX, but the
 * user implements flow control via low-level software and
 * provides a FIFO buffer—ensuring the BUSY pin is low before
 * sending each byte to the serial data line—this is treated
 * the same as having hardware flow control for the BC7215
 * driver library, and this option can be selected.
 *
 * USE_UART_INTERRUPT:
 * Refers to serial reception. Determines whether it is handled
 * by the user via serial interrupts, or if the system
 * automatically writes received data to a buffer for the user
 * to read.
 *****/

#include "bc7215.h"           // Includes BC7215 driver library, providing basic operations
#include "bc7215_ac_lib.h"    // Includes A/C Control Library header, implementing A/C control

/*****
 * Interface Functions Required by BC7215A Driver
 *****/
// Function to set MOD pin high; sets BC7215A MOD pin to high level
void set_mod_high(void) { MOD_PIN = 1; }

// Function to set MOD pin low; sets BC7215A MOD pin to low level
void set_mod_low(void) { MOD_PIN = 0; }

// Function to read MOD pin state; returns 0 or 1
uint8_t read_mod(void) { return MOD_PIN; }

/***** This section is only needed when NOT using hardware transmission flow control *****/
 * // Function to read BUSY pin state, returns 0 or 1
 * uint8_t read_busy(void)
 * {
 *     return BUSY_PIN;
 * }
 *
 * // UART single-byte send function
 * void uart_send_1_byte(uint8_t data)
 * {
 *     UART_TXD = data;    // Write data to be sent into UART TX register
 *     while (!UART_TX_COMPLETE); // Wait for data transmission to complete. Note: this is not just
 *                               // "TX Register Empty." Some MCUs show empty immediately after
 *                               // start, but data isn't fully sent yet.
 * }
 *****/

/***** This section is only needed when using hardware transmission flow control *****/
 * // UART single-byte send function
 * void uart_send_1_byte(uint8_t data)
 * {
 *     while (SEND_FIFO_IS_FULL); // If TX FIFO is full, wait
 *     write_to_send_fifo(data);  // Write data to be sent into serial TX FIFO
 * }
 *****/

/***** This section is only needed using interrupts for serial reception *****/
 * // UART interrupt enable function
 * void enable_rx_int(void)
 * {
 *     RXIE = 1;
 * }
 *
 * // UART interrupt disable function
 * void disable_rx_int(void)
 * {
 *     RXIE = 0;
 * }
 *
 * // UART RX interrupt handler
 * void uart_rx_isr(void)
 * {
 *     uint8_t data;
 *     data = UART_RXD;    // Read received data
 *     bc7215_process_uart_data(data); // Call BC7215 driver library to process
 *     // After calling bc7215_process_uart_data(), it will affect the return results of
 *     // query functions: bc7215_data_ready(), bc7215_format_ready(),
 *     // bc7215_cmd_completed(), bc7215_is_busy()
 * }
 *****/

/***** This section is only needed when using a system serial RX buffer *****/
 * // Periodically poll the serial RX buffer or driven by serial RX events
 * void check_rx_buffer(void)
 * {
 *     uint8_t data;
 *     while (RX_BUFFER_NOT_EMPTY) // Read until RX buffer is empty
 *     {
 *         data = read_from_rx_buffer(); // Read one byte
 *         bc7215_process_uart_data(data); // Call BC7215 driver library to process
 *         // After calling bc7215_process_uart_data(), it will affect the return results of
 *         // query functions: bc7215_data_ready(), bc7215_format_ready(),
 *         // bc7215_cmd_completed(), bc7215_is_busy()
 *     }
 * }

```

```

*****/

/***** Above is hardware-related code *****/
-----
*****/

/***** Below is hardware-independent code *****/

/*****
 * Variable Definitions for AC Library
 *****/
// When the AC model (data length) is uncertain, use the maximum data packet size
// as the input buffer. Usually, sampling only requires 1 data packet and
// 1 format packet buffer, but for multi-segment data formats, the signal might
// be divided into three segments. Here we use 4 to keep some margin.
bc7215DataMaxPkt_t IrRxData[4];
bc7215FormatPkt_t IrRxFormat[4];
uint8_t IrStatus[4];
bc7215DataMaxPkt_t backupBaseData; // Used to backup base data during parsing operations
uint8_t backupStatus;
bc7215CombinedMsg_t IrRxMsg[4]; // This variable is mainly used for initializing multi-segment sampled data
uint8_t sampleCount; // Number of data groups sampled

// Data to be transmitted
const bc7215DataVarPkt_t* DataToSend;
// Program control variables
bool complexModeRx; // Whether use complex mode in capture
bool formatReceived; // Whether a format packet has been received

/*****
 * Main Program
 *
 * Note: This code is for Celsius ACs only, if your AC is
 * Fahrenheit, replace corresponding functions with
 * their _f version in the code.
 *****/
main()
{
    init_uart(); // Set UART baud rate to 19200, 8 data bits, no parity, 2 stop bits

    // Setup BC7215 driver library
    bc7215_config_mod_set_high_func(set_mod_high); // Configure BC7215 driver MOD set high function
    bc7215_config_mod_set_low_func(set_mod_low); // Configure BC7215 driver MOD set low function
    bc7215_config_read_mod_func(read_mod); // Configure BC7215 driver MOD read function
    bc7215_config_uart_send_byte(send_1_byte); // Configure BC7215 driver UART single byte send function

    /***** Only used when NOT using hardware flow control *****/
    * bc7215_config_read_busy_func(read_busy); // Configure BC7215 driver BUSY read function *
    *****/

    /***** Only used when using interrupts for UART RX *****/
    * bc7215_config_uart_int_en_func(enable_rx_int); // Configure BC7215 driver UART int enable *
    * bc7215_config_uart_int_dis_func(disable_rx_int); // Configure BC7215 driver UART int disable *
    *****/

    delay(100ms); // wait for BC7215A to power up
    bc7215_set_rx(); // If BC7215A was in shutdown mode, change to RX will wake it up
    delay(50ms); // safety time between mode change
    bc7215_set_tx(); // Set BC7215A to Transmit mode on power-up
    delay(50ms);

    /***** Setup combined message pointer, it will remain unchanged through out run time *****/
    for (i = 0; i < 4; i++)
    {
        IrRxMsg[i].bitLen = 0;
        IrRxMsg[i].body.msg.fmt = &IrRxFormat[i];
        IrRxMsg[i].body.msg.datPkt = (bc7215DataVarPkt_t*)&IrRxData[i];
    }

    ac_lib_init(); // Initialize AC code library

    // Initialization complete, enter AC control or parsing state
    bc7215_load_format(bc7215_ac_get_base_fmt()); // Load base format info to BC7215A chip.
                                                // Since initialized with format previously,
                                                // can use get_base_fmt function.
                                                // If format info doesn't change, no need to reload every TX.
    while (1) // Infinite loop
    {
        working_mode_select(); // Operations that allow the user to select working mode
        if (WorkingMode_changed) // If working mode has changed
        {
            if (WorkingMode == AC_PARSING) // If switched to Parsing Mode
            {
                // Because parsing operations replace the base data packet, back it up first for restoration
                backupBaseData.bitLen = bc7215_ac_get_base_data()->bitLen;
                memcpy(backupBaseData.data, bc7215_ac_get_base_data()->data, (backupBaseData.bitLen + 7) / 8);
                backupStatus = bc7215_ac_get_base_fmt()->signature.bits.sig;
                complexModeRx = false; // In parsing mode, only simple mode is necessary
                ac_start_capture(); // Enter IR signal capture mode
            }
            else if (WorkingMode == AC_CONTROL) // If switched from Parsing Mode to Control Mode
            {
                ac_stop_capture(); // Exit signal capture mode
                bc7215_ac_replace_base(backupStatus, (bc7215DataVarPkt_t*)&backupBaseData);
                bc7215_load_format(bc7215_ac_get_base_fmt()); // Load base format packet to chip
            }
        }
        switch (WorkingMode)
        {
            case AC_CONTROL: // AC Control Mode
                get_new_setting(); // User sets new temp, mode, fan, key values
                if (AC_TempMode == Celsius) // If AC is Celsius
                {
                    DataToSend = bc7215_ac_set(temp, mode, fan, key); // Get new IR command data
                }
                else // If AC is Fahrenheit
                {
                    DataToSend = bc7215_ac_set_f(temp, mode, fan, key);
                }
            }
        }
    }
}
/*****

```

```

* If switching On/Off, call bc7215_ac_on() or bc7215_ac_off(). Returns data pointer *
* handled the same way. If On command returns NULL, no dedicated On command needed. *
*****/
UsingSpecialFormat = 0;
if (DataToSend->bitLen
    == 0) // If bitLen is 0, indicates return is actually a bc7215CombinedMsg_t* pointer
{
    bc7215_load_format(
        ((bc7215CombinedMsg_t*)DataToSend)->body.msg.fmt); // Load format info for this command
    DataToSend
        = ((bc7215CombinedMsg_t*)DataToSend)->body.msg.datPkt; // Update DataToSend to actual data
    UsingSpecialFormat = 1;
}
bc7215_IR_tx(DataToSend);
while (bc7215_is_busy()) // Check if transmission is complete
{
    delay(10ms); // Query again after 10ms
}

// Command transmission complete
if (UsingSpecialFormat == 1)
{
    bc7215_load_format(
        bc7215_ac_get_base_fmt()); // If command just sent used special format, reload base format
}
delay(100ms); // Delay 100ms to ensure sufficient interval between IR commands
break;

case AC_PARSING: // Parsing Mode
if (check_signal_captured()) // Check if a data packet has been received
{
    ac_stop_capture();
    if (AC_TempMode == Celsius) // If AC is Celsius
    {
        if (ac_parse(
            &Temp, &Mode, &Fan, &Power)) // Captured data is already in IrRxData and IrRxFormat
        {
            show_parse_result(); // Parse successful, show result
        }
        else
        {
            show_error(); // Parse failed, show error
        }
    }
    else // If AC is Fahrenheit
    {
        if (ac_parse_f(
            &Temp, &Mode, &Fan, &Power)) // Captured data is already in IrRxData and IrRxFormat
        {
            show_parse_result(); // Parse successful, show result
        }
        else
        {
            show_error(); // Parse failed, show error
        }
    }
    complexModeRx = false; // In parsing mode, only simple mode is necessary
    ac_start_capture();
}
break;
case FIND_NEXT: // Try next match
if (bc7215_ac_find_next())
{
    MatchCnt++;
    EEPROM_save_match(MatchCnt); // Save match count for future restoration
}
else
{
    show_no_more_match_message(); // Show message: No more matches
    ac_lib_init(); // Re-initialize AC library
    bc7215_load_format(bc7215_ac_get_base_fmt()); // Load base format info to BC7215A chip
}
break;
case INIT: // Re-sample initialization
sample_and_pairing();
bc7215_load_format(bc7215_ac_get_base_fmt()); // Load base format info to BC7215A chip
WorkingMode = AC_CONTROL;
break;
default:
break;
}
delay(10ms);
}
}

// Initialize AC Library
// First try to initialize using data saved in EEPROM. If that fails,
// go to execute sampling initialization.
void ac_lib_init(void)
{
    bool result;
    IrRxData[0] = EEPROM_read_data(); // Try to read saved init data, store in IrRxData[0]
    IrRxFormat[0] = EEPROM_read_format(); // Try to read saved init format info, store in IrRxFormat[0]
    IrStatus[0] = IrRxFormat[0].signature.bits.sig; // Use signature word in format data as Status
    // (Since saved data is dumped from lib, it must be
    // non-inverted data, so no need to consider inversion)
    // If AC is Celsius
    if (AC_TempMode == Celsius)
    {
        result
            = bc7215_ac_init(IrStatus[0], (bc7215DataVarPkt_t*)&IrRxMsg[0]) // Try to initialize using saved data
    }
    else // AC is Fahrenheit
    {
        result = bc7215_ac_init_f(
            IrStatus[0], (bc7215DataVarPkt_t*)&IrRxMsg[0]) // Try to initialize using saved data
    }
    if (!result) // Try to initialize using saved data
    {

```

```

        sample_and_pairing();          // If init fails, go to sampling and pairing
    }
    else // Initialization successful
    {
        MatchCnt = EEPROM_read_match(); // Read saved match count
        for (i = 0; i < MatchCnt; i++)
        {
            if (!bc7215_ac_find_next()) // If finding match fails
            {
                show_init_match_error_message(); // Show match error
                // In this case, user can decide to use the first match or re-sample init
                // Corresponding code goes here
            }
        }
    }
}

// Start Capture
void ac_start_capture(void)
{
    sampleCount = 0;
    formatReceived = false;
    bc7215_set_rx();
    delay(50ms);
    bc7215_set_rx_mode(1);
    bc7215_clr_data();
    bc7215_clr_format();
}

// Stop Capture
void ac_stop_capture(void)
{
    bc7215_set_tx();
    delay(50ms);
}

// Init (Pairing), should be called after acquiring sample data
bool ac_init(void)
{
    bool acInitOK = false;
    if (sampleCount == 1)
    {
        if (IrStatus[0] & 0x80) // check error bit
        {
            return acInitOK;
        }
        if (AC_TempMode == Celsius)
        {
            acInitOK = bc7215_ac_init(IrStatus[0], (bc7215DataVarPkt_t*)&IrRxMsg[0]);
        }
        else
        {
            acInitOK = bc7215_ac_init_f(IrStatus[0], (bc7215DataVarPkt_t*)&IrRxMsg[0]);
        }
    }
    else if (sampleCount > 1)
    {
        for (j = 0; j < sampleCount; j++)
        {
            if (IrStatus[j] & 0x80) // check error bit
            {
                return acInitOK;
            }
            if (IrStatus[j] & 0x40) // If "REV" bit is 1, invert every data byte
            {
                for (i = 0; i < (IrRxData[j].bitLen + 7) / 8; i++)
                {
                    IrRxData[j].data[i] = ~IrRxData[j].data[i];
                }
                IrRxData[j] &= 0xbf;
            }
        }
        if (AC_TempMode == Celsius)
        {
            acInitOK = bc7215_ac_init2(sampleCount, IrRxMsg, 0);
        }
        else
        {
            acInitOK = bc7215_ac_init2_f(sampleCount, IrRxMsg, 0);
        }
    }
    return acInitOK;
}

// IR Signal Parsing (Must be called after successful sampling)
bool ac_parse(int8_t* temp, int8_t* mode, int8_t* fan, int8_t* power)
{
    int i, j;
    if (sampleCount == 1)
    {
        if (!(IrStatus[0] & 0x80)) // check error bit
        {
            bc7215_ac_replace_base(IrStatus[0], (const bc7215DataVarPkt_t*)&IrRxData[0]);
        }
        else
        {
            return false;
        }
    }
    else if (sampleCount > 1)
    {
        for (j = 0; j < sampleCount; j++)
        {
            if (IrStatus[j] & 0x80) // check error bit
            {
                return false;
            }
            if (IrStatus[j] & 0x40) // If receiving status has "REV" bit set, reverse every byte of data

```

```

    {
        for (i = 0; i < (IrRxData[j].bitLen + 7) / 8; i++)
        {
            IrRxData[j].data[i] = ~IrRxData[j].data[i];
        }
        IrStatus[j] &= 0xbf;
    }
}
bc7215_ac_replace_base(sampleCount, (const bc7215DataVarPkt_t*)IrRxMsg);
}
if (AC_TempMode == Celsius)
{
    return bc7215_ac_parse(temp, mode, fan, power);
}
else
{
    return bc7215_ac_parse_f(temp, mode, fan, power);
}
}

// Check if IR signal capture is complete
bool check_signal_captured(void)
{
    if (bc7215_is_busy())
    {
        // If BC7215A is busy, update timer and return
        restart_timer();
    }
    else
    {
        if (bc7215_data_ready())
        {
            if (sampleCount < 4)
            {
                IrStatus[sampleCount] = bc7215_get_data((bc7215DataVarPkt_t*)&IrRxData[sampleCount]);
                if (bc7215_format_ready())
                {
                    formatReceived = true;
                    bc7215_get_format(&IrRxFormat[sampleCount]);
                }
                sampleCount++;
            }
            restart_timer();
        }

        if (sampleCount > 0)
        {
            // If sampled data has been received, end sampling after 200ms idle
            if (timer > 200ms)
            {
                return true; // Sampling complete
            }
        }
    }
    return false;
}

// Sampling and Pairing. Designed here to return only when pairing is successful.
void sample_and_pairing(void)
{
    while (
        1) // Designed as infinite loop until initialization succeeds. Consider adding forced exit in practice.
    {
        show_init_message(); // Show prompt: Remind user to set remote to Cool mode 25°C and press Fan key
        complexModeRx = true; // Use complex mode in pairing
        ac_start_capture(); // Start sampling

        // ***** Receive IR Signal (Signal Capture) *****
        while (1) // Infinite loop, consider adding forced exit in practice
        {
            if (check_signal_captured()) // Wait for sampling to end
            {
                ac_stop_capture();

                if (formatReceived)
                {
                    if (ac_init()) // Sampling results are already in IrRxData and IrRxFormat
                    {
                        EEPROM_save_data();
                        EEPROM_save_format();
                        EEPROM_save_match(0);
                        show_ok_message(); // Pairing successful
                        break;
                    }
                    else // Initialization failed
                    {
                        show_err_message(); // Sampling/Pairing failed. Remind user to check remote settings and
                        // retry.
                    }
                }
            }
            delay(10ms);
        }
    }
}
}

```